Instalación de programas en Linux

# Introducción

Clásicamente, los programas y aplicaciones en Linux (y sistemas Unix en general) se instalan compilando del código fuente. Un programa, se compone, principalmente, de:

* Ejecutables
* Librerías
* Archivos de configuración
* Manuales
* Documentos
* Otros recursos

Estos componentes del programa se guardan en un árbol de directorio con una estructura definida

* Ejecutables se guardan en el directorio bin/
* Librerías se guardan en el directorio lib/
* Archivos de configuración se guardan en etc/
* Manuales se guardan en share/man
* Documentos se guardan en share/doc
* Otros recursos se guardan en share/NOMBRE\_APLICACION

Esto es una estructura general, y puede que cada aplicación cree directorios extra dentro de cada uno de bin, lib, etc, share. Por ejemplo, los accesos directos de las aplicaciones se encuentran en /share/applications (archivos .desktop)

En general, el proceso de compilar un programa es “engorroso”: Los programas ***“dependen”*** de otros programas. Ejemplo:

* “quiero instalar python 2.7”
* “pero antes de instalarlo, necesito la librería estándar del lenguaje C, porque python fue escrito en C”
* “pero además, python ofrece un intérprete, el cual usa la terminal, hay que instalar la librería del terminal”
* “el intérprete de python también tiene ciertas características de terminal avanzadas, que las provee una librería llamada ncurses”
* “python tiene una base de datos interna”
* “python se conecta a internet”
* “python esto, python aquello”

La cosa es que instalar python (o cualquier programa) no es trivial.

# Las distribuciones de Linux solucionan estos problemas

Por ejemplo, en el caso de python:

* “La librería del C es parte de la distribución del sistema operativo”
* “Todos los programas usan características del terminal, por lo tanto, también es parte del SO”
* “ncurses será parte del SO, porque aplicaciones internas propias de la distribución también lo necesitan”
* “hay tantos programas que se conectan a internet, que se ha decidido que las librerías necesarias que hacen posible esta característica sean agregadas a la distribución”

Entonces, ¿puedo ahora instalar “python”?

Distribuciones antiguas: Sí, por defecto tenemos instalados el *código fuente*, *librerías* y *cabeceras* de todos esos programas en nuestra distribución para que puedas compilar estos programas.

Distribuciones nuevas: Sí, te ofrecemos la opción de descargar “python” compilado. Si quieres compilarlo tú, puedes bajar las *librerías ya compiladas* y *archivos de cabecera necesarios* para que la compilación sea exitosa.

En las distribuciones de Linux modernas, existe un sistema de instalación de software que se llama “gestión de paquetes de software” (software package management). Este sistema, a grandes rasgos se encarga de lo siguiente:

* permite la instalación de programas ya compilados (binarios)
* permite la instalación de aquellos programas y librerías del cual el programa a instalar “depende”
* asegura la consistencia del sistema, procurando “la compatibilidad entre los programas de acuerdo con la versión”
* permite la desinstalación de los programas sin “quebrar el sistema”, o sea, no desinstalar aquello que aún es requerido por otros programas.
* otras magias negras (ej. Corregir un sistema corrompido)

Sistemas famosos de gestión de paquetes de software:

* dpkg (DEBIAN, UBUNTU), archivos .deb
* rpm (REDHAT, SUSE), archivos .rpm
* pkg (FREEBSD), archivos .txz
* pacman (ARCHLINUX), archivos pkg.tar..xz

En el curso nos centraremos más en dpkg (Debian), aunque una vez entendiendo este sistema, es más fácil entender rpm.

## Un archivo .deb

Un archivo .deb es un archivo comprimido que en su interior contiene tres archivos:

* debian-binary, un archivo de texto que en su interior contiene la versión del formato del paquete de debian.
* control.tar[.gz|.xz], archivo que contiene un archivo llamado “control” con la información del paquete, y otro archivos:
* “md5sums” con las sumas md5 de todos los archivos incluídos.
* conffiles, lista de archivos que el paquete considera como “archivos de configuración”
* preinst, script a ejecutar justo antes de la instalación
* postinst, script a ejecutar después de la instalación
* pererm, script a ejecutar antes de la desinstalación
* postrm, script a ejecutar después de la desinstalación
* shlibs o symbols, dependencias de librerías compartidas.
* data.tar[.gz|.xz], archivo que contiene los archivos a copiar

Ejemplo de archivo .deb:

gnutls-bin\_3.5.18-1ubuntu1.2\_i386.deb

* nombre del paquete: **gnutls**-**bin**
* versión: **3.5.18**
* número de revisión: **1ubuntu1.2**
* arquitectura: **i386** (otras arquitecturas: amd64, arm64, all, etc)

Para instalar un paquete .deb, se utiliza “dpkg -i archivo.deb”. Sin embargo, dpkg instalará el archivo sólo si se encuentran las dependencias instaladas.

Para desinstalar un paquete .deb, se utiliza “dpkg -r nombre\_del\_paquete. Si hay paquetes instalados que dependen del que se va a desinstalar, entonces dpkg no hara la desinstalación del paquete. La opción “-r” no elimina los archivos de configuración del paquete (los archivos descritos en “conffiles”). Para asegurar que se elimine todo, en lugar de -r se usa la opción -P (purgar).

## Dependencias

Si bien instalar paquetes usando el administrador de paquetes nos ayuda mantener la consistencia del sistema operativo, sigue siendo engorroso, porque cada paquete tiene “dependencias”, esto es, un paquete requiere que otros paquetes estén instalados para funcionar. Este problema es análogo al problema de instalar programas compilados. Para solucionar esto, las distribuciones de Linux (y otros sistemas operativos) ofrecen **sistemas de resolución de dependencias**.

├─ python2.7

├─ libpython2.7-stdlib

│ ├─ libbz2-1.0

| | └─ libc6 (>= 2.4)

| | └─ libgcc1

| | ├─ gcc-8-base (= 8.3.0-6ubuntu1~18.04.1)

| | └─ libc6 (>= 2.14)

│ ├─ libc6 (>= 2.15)

│ ├─ libdb5.3

| | └─ libc6 (>= 2.17)

│ ├─ libexpat1 (>= 2.1~beta3)

| | └─ libc6 (>= 2.25)

│ ├─ libffi6 (>= 3.0.4)

| | └─ libc6 (>= 2.14)

│ ├─ libncursesw5 (>= 6)

| | ├─ libc6 (>= 2.14)

| | └─ libtinfo5 (= 6.1-1ubuntu1.18.04)

| | └─ libc6 (>= 2.16)

│ ├─ libpython2.7-minimal (= 2.7.17-1~18.04)

│ ├─ libreadline7 (>= 7.0~beta)

| | ├─ libc6 (>= 2.14)

| | ├─ libtinfo5 (= 6.1-1ubuntu1.18.04)

| | └─ readline-common

| | ├─ dpkg (>= 1.15.4)

| | | ├─ tar (>= 1.28-1)

| | | | └─ libselinux1 (>= 1.32)

| | | ├─ libacl1 (>= 2.2.51-8)

| | | ├─ libc6 (>= 2.17)

| | | | ├─ libbz2-1.0

| | | | ├─ libc6 (>= 2.14)

| | | | ├─ liblzma5 (>= 5.2.2)

| | | | └─ libc6 (>= 2.17)

| | | ├─ libselinux1 (>= 2.3)

| | | | ├─ libc6 (>= 2.14)

| | | | └─ libpcre3

| | | | └─ libc6 (>= 2.14)

| | | ├─ libzstd1 (>= 1.3.2)

| | | └─ zlib1g (>= 1:1.1.4)

| | └─ install-info

| | ├─ libc6 (>= 2.14)

| | └─ dpkg (>= 1.16.1)

│ ├─ libsqlite3-0 (>= 3.5.9)

| | └─ libc6 (>= 2.14)

│ ├─ libssl1.1 (>= 1.1.1)

| | ├─ debconf (>= 0.5)

| | | └─ perl-base (>= 5.20.1-3~)

| | | ├─ dpkg (>= 1.17.17)

| | | └─ libc6 (>= 2.23)

| | ├─ debconf-2.0

| | └─ libc6 (>= 2.25)

│ ├─ libtinfo5 (>= 6)

│ └─ mime-support

├─ mime-support

└─ python2.7-minimal (= 2.7.17-1~18.04)

├─ libpython2.7-minimal (= 2.7.17-1~18.04)

├─ libc6 (>= 2.15)

└─ zlib1g (>= 1:1.2.0)

Lo anterior es un ejemplo de dependencias de paquetes del paquete “python2.7”. Sin un sistema de resolución de dependencias, para poder instalar el paquete “python2.7” con dpkg tendríamos que consultar la documentación de Debian, Ubuntu, o del sistema operativo en cuestión, para tener la información de las dependencias, luego, descargarlas, (fijándose bien en las versiones), y finalmente, instalarlas en el orden que corresponde, partiendo por el más necesario primero, que en este caso, el más básico es “libc6” (la librería del C).

Para evitarnos este problema, los sistemas de resolución de dependencias hacen este trabajo automático. De hecho, el ejemplo anterior se generó usando el programa “apt-rdepends” de la suite “apt”, aunque la salida está arreglada para ilustrar mejor el “árbol de dependencias”.

Algunos sistemas de resolución de dependencias por sistema operativo:

* Debian/Ubuntu y derivados: apt, apt-get, aptitude
* Redhat, Centos, Fedora y derivados: yum, dnf (yum mejorado)
* Suse Linux: zypper
* Freebsd: pkg
* Archlinux: pacman

## Apt

Algunos programas:

* apt-get: instalación, actualización y eliminación de software
* apt-cache: manipulación de información de repositorios
* apt-rdepends: visualización de dependencias de paquetes
* apt: interfaz para utilidades apt
* aptitude: interfaz para utilidades apt

Instalar/actualizar programas:

* apt-get install NOMBRE\_PAQUETE
* apt install NOMBRE\_PAQUETE
* aptitude install NOMBRE\_PAQUETE

Desinstalar programas:

* apt-get remove NOMBRE\_PAQUETE
* apt-get purge NOMBRE\_PAQUETE
* apt remove NOMBRE\_PAQUETE
* apt purge NOMBRE\_PAQUETE
* aptitude remove NOMBRE\_PAQUETE
* aptitude purge NOMBRE\_PAQUETE

Buscar paquetes por descripción o palabra clave

* apt-cache search PALABRA\_O\_PATRON
* apt search PALABRA\_O\_PATRON

### Repositorios

Los paquetes que apt instala son obtenidos de lo que es llamado una “origen de software” (software sources), o también llamados repositorios (software repositories). Ocurre lo mismo en otros sistemas de paquetes de otros sistemas operativos. Los respositorios Debian (el tipo de repositorio que también utiliza Ubuntu) son un conjunto de paquetes binarios o de fuentes organizados en un árbol especial de directorios, además de archivos de infraestructura (checksums, índices, firmas, descripciones, traducciones, etc). Los repositorios pueden encontrarse en un servidor web, una ubicación física (disco duro, cd, dvd, etc), o una ubicación en red (nfs, samba, etc).

El sistema apt guarda información sobre qué repositorios utilizar en los archivos /etc/apt/sources.list y /etc/apt/sources.list.d/ARCHIVOX.list

**deb http://archive.ubuntu.com/buntu/ bionic main restricted**

# deb-src http://archive.ubuntu.com/buntu/ bionic main restricted

**deb http://archive.ubuntu.com/buntu/ bionic-updates main restricted**

# deb-src http://archive.ubuntu.com/buntu/ bionic-updates main restricted

**deb http://archive.ubuntu.com/buntu/ bionic universe**

# deb-src http://archive.ubuntu.com/buntu/ bionic universe

deb http://archive.ubuntu.com/buntu/ bionic-updates universe

# deb-src http://archive.ubuntu.com/buntu/ bionic-updates universe

deb http://archive.ubuntu.com/buntu/ bionic multiverse

# deb-src http://archive.ubuntu.com/buntu/ bionic multiverse

deb http://archive.ubuntu.com/buntu/ bionic-updates multiverse

# deb-src http://archive.ubuntu.com/buntu/ bionic-updates multiverse

deb http://archive.ubuntu.com/buntu/ bionic-backports main restricted universe multiverse

# deb-src http://archive.ubuntu.com/buntu/ bionic-backports main restricted universe multiverse

# deb http://archive.canonical.com/buntu bionic partner

# deb-src http://archive.canonical.com/buntu bionic partner

deb http://security.ubuntu.com/buntu/ bionic-security main restricted

# deb-src http://security.ubuntu.com/buntu/ bionic-security main restricted

deb http://security.ubuntu.com/buntu/ bionic-security universe

# deb-src http://security.ubuntu.com/buntu/ bionic-security universe

deb http://security.ubuntu.com/buntu/ bionic-security multiverse

# deb-src http://security.ubuntu.com/buntu/ bionic-security multiverse

Lo de arriba es un ejemplo del archivo /etc/apt/sources.list, el cual contiene los orígenes del software principales del sistema operativo. El formato de este archivo es el siguiente:

* tipo de paquete: deb son paquetes binarios, deb-src son paquetes que contienen código fuente
* Ubicación: un URI que apunta a la ubicación donde encontrar los archivos. En este caso, de un servidor web (http://)
* Nombre de la distribución: las distribuciones de Ubuntu tienen nombres como “Edgy”, “hefty”, “gutsy”, “bionic”. En el ejemplo se muestra la distribución “bionic”. Además se ofrecen repositorios especiales “bionic-updates”, “bionic-backports” y “bionic-security”.
* Todo lo que venga después son secciones dentro del repositorio para clasificar software, por ejemplo “main” contiene paquetes fundamentales, “restricted” son paquetes de software cuyo código fuente no es posible obtenerlo pues es privativo, “universe” es repositorio de software que tiene soporte por parte de Canonical, la empresa detrás de Ubuntu. “multiverse” son paquetes de software libre a los cuales no se les ofrece soporte oficial.

Cada línea se transforma en un URL de donde se sacarán los archivos. Revise la URL <http://archive.ubuntu.com/ubuntu/>, especialmente los directorios dists, pool y índices.

### Instalar software

El procedimiento para instalar software es el siguiente (como root, o con sudo):

1. Ejecutar “apt update”: este paso actualiza la información sobre los repositorios en el computador. Una vez ejecutado, el sistema operativo sabrá qué software puede ser actualizado, y las versiones disponibles del software nuevo.
2. Ejecutar “apt install PAQUETE\_DE\_SOFTWARE”: Este paso prepara al sistema para instalar software. Apt resuelve la dependencia de los paquetes y luego mostrará un resumen de los cambios a realizar. Puede que ocurra que:
   1. El sistema tenga que eliminar paquetes que hagan conflicto, y demás paquetes que dependan de él
   2. El sistema tenga que actualizar paquetes que ya están instalados
   3. El sistema sugiere y recomienda paquetes adicionales (hay opciones para instalar automáticamente estos paquetes)
   4. El sistema tenga que instalar paquetes adicionales para resolver dependencias.
3. Aceptar la propuesta de apt. Esto instalará el software nuevo, y realizará los cambios en el sistema, como ejecutar scripts adicionales, iniciar servicios, o incluso, preguntar al usuario sobre cambios que tengan que hacerse a archivos de configuración.
4. Verificar el software instalado. Puede intentar usar “dpkg -L NOMBRE\_PAQUETE” para ver los archivos instalados. Por lo general, las aplicaciones son instaladas en el árbol de directorios bajo /usr (/usr/bin, /usr/lib, /usr/include, etc).

### Actualizar software

1. Ejecutar “apt update”. Si hay actualizaciones disponibles, el programa dirá un mensaje para revisar los paquetes que se pueden actualizar con “apt list –upgradable”
2. Ejecutar “apt upgrade”

Se debe tener en cuenta que las actualizaciones pueden provocar que algún software deje de funcionar, debido al cambio de sus funciones internas. Se debe verificar con la documentación si alguna actualización puede ser perjudicial.

### Eliminar software

1. Ejecutar “apt remove NOMBRE\_DE\_PAQUETE” para eliminar software pero mantener los archivos de configuración
2. Ejecutar “apt purge NOMBRE\_DE\_PAQUETE” para eliminar software y también eliminar los archivos de configuración (sólo los indicados en conffiles)

## Aplicaciones, librerías y desarrollo

Muchas aplicaciones dependen de “librerías” para funcionar. ¿Qué son las librerías? Son archivos que tienen funciones programadas que son utilizadas por otros programas. Por ejemplo, si un programa está escrito en Python, utilizando Numpy, entonces el programa requerirá Python para ser ejecutado y Numpy instalado para que las funciones que utilizan a Numpy pueda funcionar. Lo mismo ocurre con todos los programas. Por ejemplo existe un programa llamado “curl” para transferir archivos remotamente usando muchos protocolos (http, https, ftp, etc). El comando es curl, pero todas las funciones para transferir archivos no se encuentran físicamente dentro del comando “curl”, sino que el comando es una interfaz hacia la librería “libcurl”, y la librería tiene todas las funciones para descargar archivos de internet. ¿Por qué se separan librerías y programas? Porque de esta forma, muchos programas que necesiten la capacidad de descargar archivos de internet podrán tener esta capacidad sin necesidad de reprogramarla, sino que pueden “usar” las funciones que provee la librería. Muchos programas en Linux que requieren descargar de internet fueron programadas para utilizar libcurl, por lo tanto, libcurl es una dependencia de muchos programas del sistema operativo.

~$ apt depends curl

curl

Depends: libc6

Depends: libcurl4

Depends: zlib1g

~$ apt depends libcurl4

libcurl4

Depends: libc6

Depends: libgssapi-krb5-2

Depends: libidn2-0

Depends: libldap-2.4-2

Depends: libnghttp2-14

Depends: libpsl5

Depends: librtmp1

Depends: libssl1.1

Depends: zlib1g

Aquí se puede ver que el programa “curl” depende del paquete de librerías “libcurl4”, el cual provee las funciones necesarias para que curl funcione. Otros programas pueden tener la misma dependencia.

Ahora vemos los contenidos de ambos paquetes:

~$ dpkg -L curl

/.

/usr

/usr/bin

**/usr/bin/curl**

/usr/share

...

~$ dpkg -L libcurl4

/.

/usr

/usr/lib

/usr/lib/x86\_64-linux-gnu

**/usr/lib/x86\_64-linux-gnu/libcurl.so.4.5.0**

/usr/share

...

**/usr/lib/x86\_64-linux-gnu/libcurl.so.4**

El paquete “curl” provee el ejecutable (archivo en /usr/bin). El paquete libcurl provee la “librería de ejecución”, o “runtime”, o sea, la librería que los programas utilizan al momento de ejecutar. En los sistemas Unix, una librería de entorno de ejecución es un “shared object” o “objeto compartido”, o “librería compartida”, o también llamado “librería dinámica”, y normalmente el archivo tendrá el formato “libXXX.so.VERSION”.

Un tercer tipo de paquete es el de librería de desarrollo, conocido como paquetes “dev”, “devel” o “developtment”. Estos paquetes traen lo necesario para poder “compilar” un código fuente que utiliza librerías externas. En debian y Ubuntu, estos paquetes tienen el formato de nombre “libXXX-dev”. El contenido de estos paquetes difieren a un código fuente en que traen librerías ya compiladas, pero no enlazadas, y traen archivos que ayudan a la compilación de programas que utilizan las funciones provistas por las librerías (headers e includes).

~$ dpkg -L libcurl4-gnutls-dev

/.

/usr

/usr/bin

/usr/bin/curl-config

/usr/include

/usr/include/x86\_64-linux-gnu

/usr/include/x86\_64-linux-gnu/curl

/usr/include/x86\_64-linux-gnu/curl/curl.h

/usr/include/x86\_64-linux-gnu/curl/curlver.h

/usr/include/x86\_64-linux-gnu/curl/easy.h

/usr/include/x86\_64-linux-gnu/curl/mprintf.h

/usr/include/x86\_64-linux-gnu/curl/multi.h

/usr/include/x86\_64-linux-gnu/curl/stdcheaders.h

/usr/include/x86\_64-linux-gnu/curl/system.h

/usr/include/x86\_64-linux-gnu/curl/typecheck-gcc.h

/usr/lib

/usr/lib/x86\_64-linux-gnu

/usr/lib/x86\_64-linux-gnu/libcurl-gnutls.a

/usr/lib/x86\_64-linux-gnu/libcurl-gnutls.la

/usr/lib/x86\_64-linux-gnu/pkgconfig

/usr/lib/x86\_64-linux-gnu/pkgconfig/libcurl.pc

/usr/share

...

/usr/lib/x86\_64-linux-gnu/libcurl-gnutls.so

/usr/lib/x86\_64-linux-gnu/libcurl.a

/usr/lib/x86\_64-linux-gnu/libcurl.la

/usr/lib/x86\_64-linux-gnu/libcurl.so

...

# Instalación de software a partir de código fuente

Muchas veces, será necesario instalar software a partir del código fuente. Se puede llegar a esta decisión a partir de algunos motivos:

* necesitas una versión del software más nueva (o más vieja incluso) que la que ofrece el sistema operativo
* El sistema operativo no ofrece opción dentro de su repositorio
* Las versiones ya compiladas (ya sea ofrecida en el repositorio del SO, u otro desde internet) carece de alguna característica particular, la cual se puede activar al compilar
* No tienes privilegios de súper usuario, entonces instalar desde el código fuente es una opción.

La decisión de instalar software tiene ciertas implicancias, la principal es la dependencia de otros programas y/o librerías, pero además hay que tener en cuenta:

* necesitarás las aplicaciones destinadas a compilar e instalar software desde el código fuente
  + En sistemas debian, esto se soluciona con “apt-get install build-essential”
* deberás tener especial cuidado de no romper la consistencia del sistema operativo: esto se traduce a la regla “NO INSTALAR SOFTWARE EN /usr” (obviamente a menos que sepa lo que está haciendo)
* instalar software desde el código fuente no es trivial, y requiere desarrollar ciertas habilidades que sólo lo dará la experiencia. Habrá que entender algunos archivos y el proceso de compilación para entender la causa de algunos errores en la compilación.

### Programas necesarios para instalar desde el código fuente

Esto dependerá del software a instalar, pero en general necesitarás:

* el compilador: en Linux, el más usado es GCC (son muchos compiladores), pero hay más opciones, como el compilador de Intel (ICC)
* Make: es un programa que asiste en la compilación de código fuente complejo. el comando es “make”, y el archivo principal a tener en cuenta es “Makefile”
* Otros programas pueden ser necesarios, dependiendo del software a instalar, por ejemplo “pkg-config”

### Una distinción entre lenguajes de programación “compilados” y lenguajes “interpretados”

Un **lenguaje de programación compilado** es aquel que, antes de poder ejecutar las instrucciones programadas, éstas deben ser compiladas a código de máquina (lenguaje del CPU). Esto ofrece ventajas y desventajas. La ventaja, es que la ejecución de las instrucciones es rápida, porque el código a ejecutar ya no necesita ser traducido al lenguaje de máquina, es nativo. Otra ventaja es que el compilador verifica que el código no tenga errores de sintaxis antes de compilar y la compilación resuelve todas las dependencias a otras librerías, entonces se puede asegurar que el código a ejecutar está “bien escrito” (puede tener errores lógicos, claro). La desventaja es que el proceso de compilar puede ser lento y tedioso dependiendo del tamaño de proyecto, y al momento de querer encontrar errores, hay que recompilar para verificar las correcciones. Ejemplos de lenguajes compilados: C, C++, Objective C, Java, Fortran, Scala.

Un lenguaje de programación interpretado es aquel que es ejecutado directamente desde el código fuente. La traducción al código de máquina es realizada “en vivo y en directo” durante la ejecución. Normalmente, no se compila a código de máquina directamente, sino a un lenguaje intermedio (llamado bytecode) que es interpretado por una máquina virtual que ya está cargada en memoria al momento de ejecutar. La ventaja de estos lenguajes es la rapidez al momento de programar y realizar pruebas, ya que no es necesaria la compilación. La desventaja es que en general, la ejecución es mucho más lenta que la ejecución de un programa compilado, y puede haber errores escondidos en partes del código que no son ejecutadas frecuentemente. Ejemplos de lenguajes interpretados: Python, PHP, Perl, Ruby.

## El proceso de compilar un programa

Un compilador es un programa que convierte un código de un tipo a un código de otro tipo. El código a convertir es llamado el “código fuente”, y el código convertido es llamado “código objetivo” (source code y target code en inglés). El código fuente suele ser en un lenguaje del tipo “alto nivel”, esto es “legible y entendible para el humano”, y el código objetivo suele ser “lenguaje de máquina”, o sea, un archivo binario con instrucciones que entiende el procesador del sistema (el CPU).

código fuente

código de máquina

### Caso muy frecuente: programa escrito en C

El código fuente en C se compone principalmente de:

* archivos .h, archivos de cabecera, contiene declaraciones de variables, funciones, constantes. A estos archivos se les conoce como “headers” o “includes”
* archivos .c, programa escrito en c, contiene definición de variables y funciones, o sea implementación. Estos son principalmente el código fuente, y hacen referencia a archivos de cabeceras (ya sea propio del programa a compilar, u otros provistos por otro software)

En un sistema con Linux, el compilador por defecto es el GCC (GNU Compiler Collection), que incluye compiladores para C (gcc), C++ (g++), y Fortran (gfortran). Para compilar un programa en C, el comando es:

gcc -o ARCHIVO\_RESULTANTE archivo\_fuente.c

Los actores en este proceso son:

* el compilador (gcc), lee el código en archivo\_fuente.c y reemplaza las líneas que dicen “#include <archivo.h>” por el archivo .h correspondiente. También revisa otras directivas de preprocesador (líneas que comienzan con #) y hace los reemplazos correspondientes. A este proceso se le llama “preprocesamiento”
* el compilador luego procede a realizar efectivamente la compilación, traduce el código escrito en C por código en lenguaje ensamblador.
* Luego, gcc invoca al ensamblador (as) transforma el código en lenguaje ensamblador en un archivo binario en lenguaje de máquina. El contenido del archivo lo entiende el procesador (CPU).
* Luego, gcc invoca al enlazador o linker (ld). El archivo a compilar, si utiliza librerías externas, éstas deben ser incluidas en el binario resultante. Para lograr esto, en enlazador puede realizar una de dos cosas:
  + Enlazar estáticamente, o sea, copia el código binario de la librería y lo inserta en el ejecutable resultante
  + Enlazar dinámicamente, o sea, en el ejecutable resultante quedará una “referencia” al archivo que provee las funciones necesarias. Al momento de ejecutar el programa, el archivo deberá existir en las ubicaciones que el sistema operativo reconozca como ruta para enlace dinámico (o las que estén en la variable de entorno LD\_LIBRARY\_PATH).

Con gcc entonces podemos crear librerías estáticas, dinámicas y ejecutables a partir del código fuente.

Nueva Librería .so

Nuevo ejecutable

Archivos .c

código de máquina

Archivos .h

Librería .so

Librería .a

Archivos .h

* Compilar código fuente a código de máquina, sin enlazar (-c)

gcc -c -o maquina.o fuente.c

* Compilar código fuente a código de máquina, para ser enlazado dinámicamente

gcc -c -fPIC -o futuralibreria.o futuralibreria.c

* Enlazar código máquina estáticamente para crear ejecutable independiente:

gcc -o ejecutable ejecutable.o

# o también gcc -o ejecutable ejecutable.c

* + (ejecutable.c debe tener la función main() )
* Crear librería estática para crear otros ejecutables

ar rcs libreria1.o libreria2.o libreria3.o libreria.a

* Enlazar código máquina dinámicamente para crear librería dinámica

gcc -shared libreria1.o libreria2.o -o libreria.so

* + (los archivos .o debieron ser compilados con la opción -fPIC para que esto funcione)
* Compilar y enlazar ejecutable con librería dinámica

gcc ejecutable.c -o ejecutable -L/ruta/librería -lreria

# OJO, el orden es importante! Primero compila y luego enlaza, por lo tanto, las librerías a enlazar deben estar al final.

# la opción -l toma como argumento el nombre de la librería. Si el archivo se llama “libalgo.so” entonces el nombre de la librería es “algo” y la opción se debe escribir “-lalgo”.

En estos casos se puede ver que el comando “gcc” es usado como compilador y como enlazador. Gcc reconoce cuando se usa como enlazador cuando le entregan como archivo de entrada un objeto ya compilado, y automáticamente invoca a “ld”. “gcc -c” sólo crea el objeto compilado sin realizar el enlace.

Además, cuando gcc es compilador, debe conocer la ruta a los archivos “.h” que vienen con el código fuente. Las rutas ya conocidas son las de /usr/include y /usr/local/include. Otras rutas deberán ser proporcionadas por la opción -I/ruta

gcc -o ejecutable -I/ruta/a/directorio/con/archivos\_h codigofuente.c

Lo mismo con las librerías, cuando gcc es también enlazador, se le debe entregar la ruta a las librerías, y especificar los nombres de las librerías a enlazar. La opción “-L” proporciona la ruta, y “-l” proporciona el nombre de la librería. Por defecto, las librerías están en /usr/lib, /usr/lib64.

gcc -o ejecutable -L/ruta/a/librerias -lnombre\_libreria codigofuente.c

# en este caso, el archivo de la librería deberá llamarse “libnombre\_libreria.a” o “libnombre\_libreria.so” (para enlace dinámico)

Sugiero practicar con esta guía sencilla:

<https://renenyffenegger.ch/notes/development/languages/C-C-plus-plus/GCC/create-libraries/index>

## Compilar código fuente de programas

Como decía antes, puede haber casos en que sea necesario instalar programas desde el código fuente.

Podemos tomar como ejemplo el código fuente de Python: <https://www.python.org/ftp/python/3.8.1/Python-3.8.1.tgz>

1. Descargar y descomprimir el archivo con tar -zf
2. Revisar el contenido del directorio creado. Los archivos importantes (entre varios) son:
   1. INSTALL (indicaciones para instalar)
   2. README (información sobre el programa)
   3. configure (script ejecutable que inspecciona el sistema operativo para determinar si se podrá realizar la compilación, y prepara los archivos en el directorio para ser compilados según lo que encontró con la inspección)
   4. Makefile.in (archivo prototipo que usará configure para crear a Makefile, el cual contiene todos los comandos necesarios para compilar todos el código fuente. Este archivo puede estar presente en muchos subdirectorios, pero el que está en el nivel más alto es el principal).
3. Ejecutar “./configure --help"
   1. La opción --prefix sirve para indicar dónde instalar el programa
   2. Las opciones con nombre --enable-XXX (o --disable-XXX) son para activar o desactivar características del programa
   3. Las opciones con nombre --with-XXX (o --without-XXX, o --with-XXX=/ruta/) son para indicar a configure donde encontrar ciertas librerías externas requeridar por las características, o para especificar versiones específicas de las librerías externas, en caso de que haya múltiples instaladas.
4. Después de tomar las decisiones correspondientes, ejecutar ./configure, con las opciones escogidas (siendo prefix una de las más importantes, ya que puedes instalar software en tu home sin requerir permisos de root).
5. Si configure falló
   1. Muy probable que el error sea porque configure no halló una librería. Puede que el error indique “No encontré archivox.h”, “No encuentro algolib (o lib-algo)” o incluso alguno configure podrían sugerir que falta un paquete instalado y cómo instalarlo según el sistema operativo. En este caso, habrá que buscar qué paquete de desarrollo provee el archivo que falta. Para eso, verificar la documentación del paquete (INSTALL), o buscar en packages.ubuntu.com, o al sitio que corresponda dependiendo del sistema operativo. También se puede usar “apt search” e indicar el nombre de la librería.
   2. Otra causa puede ser que la versión de una librería instalada no sea compatible con la requerida. En este caso configure dirá algo como “libalgo: 2.3 found, >=2.5 needed”, o sea, encontró la versión 2.3, pero debe estar la 2.5 o superior. En este caso, es recomendable buscar la librería e instalarla separada del sistema operativo (en un directorio cualquiera, menos de sistema), y con configure indicar “--with-libalgo=/ruta/a/nuevolibalgo”.
   3. Otra causa puede ser “C compiler cannot create executables”, esto quiere decir que hay un problema con gcc (o el compilador utilizado). Verificar cargando gcc a secas, puede ser que no esté instalado.
6. Si configure fue exitoso, el siguiente paso es ejecutar “make”. El comando make abrirá Makefile y ejecutará ese script, el cual a su vez ejecuta todo lo necesario para realizar la compilación.
7. Si “make” falló:
   1. Encontrar la causa de la falla en “make” no es trivial: normalmente el error que se muestra corresponde al comando que falló dentro de make: puede ser el compilador o el linker. La mayoría de las veces es el linker.
   2. Cuando es el linker, los errores dirán cosas como “undefined reference to …”.
   3. Cuando es el compilador, dirá errores como “archivox.h not found”, o “Syntax error”. Estos errores se pueden corregir usando algunas variables de entorno, las cuales se explican más abajo.
   4. Si la falla tenía que ver con sintáxis en un archivo de código fuente, se puede corregir, y luego ejecutar “make” nuevamente. El proceso continuará de donde quedó.
8. Si “make” es exitoso, ejecutar “make install”
9. Si make install falló, muchas veces es por falla en los permisos. Si en configure “--prefix” apunta a una ubicación de sistema (por ejemplo, /usr/local), entonces “make install” debe ser ejecutado como superusuario. Otra causa de falla, es que algún archivo no existe; en este caso, hay que verificar nuevamente “make”, porque puede que haya finalizado con error, o que falte enlazar algún archivo.
10. Si “make install” fue exitoso, verificar “prefix” (el directorio).

Los paquetes de código fuente estándar crean un árbol de directorios del tipo unix dentro del directorio indicado por el prefix. Estos incluyen a bin, lib, etc, include, share, y otros. Hay que recordar que si se instaló una aplicación, hay que agregar ese nuevo directorio bin a la variable de entorno PATH:

export PATH=/nuevo/directorio/bin:$PATH

Si la instalación incluye un directorio “lib”, entonces debemos agregarlo a la variable “LD\_LIBRARY\_PATH” si contiene archivos “.so”, y a “LIBRARY\_PATH” si contiene archivos “.a”

export LD\_LIBRARY\_PATH=/nuevo/directorio/lib:$LD\_LIBRARY\_PATH

export LIBRARY\_PATH=/nuevo/directorio/lib:$LIBRARY\_PATH

#LD\_LIBRARY\_PATH en MacOS se llama DYLD\_LIBRARY\_PATH (pero se recomienda el uso de DYLD\_FALLBACK\_LIBRARY\_PATH para evitar conflictos)

### Variables de entorno al momento de compilar

Cuando ejecutamos “configure”, éste captura algunas variables de entorno que luego las incorpora en “Makefile”. Cambiar éstas variables puede ayudar a corregir errores. Cada vez que se realiza cambio a las variables, habrá que ejecutar “configure” nuevamente para que éstas sean capturadas. Antes de ejecutar “configure” nuevamente, si ya se había comenzado a compilar, se debe ejecutar:

* make clean: elimina los archivos generados en la compilación
* make distclean: elimina los archivos generados en la configuración

Las variables son:

* CC: el compilador de C
* CXX: el compilador de C++
* FC, F90, F77: el compilador de fortran
* CFLAGS: opciones para el compilador del C
* CXXFLAGS: opciones para el compilador del C++
* FCFLAGS, F90FLAGS, F77FLAGS: opciones para el compilador de fortran
* CPPFLAGS: opciones para el preprocesador de C y C++ (todo lo referente a los archivos .h y defines)
* LDFLAGS: opciones para el enlazador (todo lo referente a las librerías)

Estas variables se pueden definir antes de configure usando “export”, o en la misma línea. Ejemplo:

$ CC=$HOME/bin/gcc CXX=$HOME/bin/g++ CFLAGS=”-std=c99 -fPIC” CPPFLAGS=”-I$HOME/include” LDFLAGS=”-L$HOME/lib -L/$HOME/otro/lib” ./configure --prefix= $HOME/software

Dependerá de cada paquete de software cuáles opciones son leídas. Para saberlo, ejecutar “./configure --help”, donde encontrará las variables leídas y una explicación.

Entonces, si se encontró errores en la compilación o enlace, puede ser que falte indicar opciones al preprocesador y enlazador, o sea, CPPFLAGS y LDFLAGS respectivamente. También puede ser que haya que agregar opciones a CFLAGS (o al compilador que corresponda) que tengan que ver con el estándar a usar para compilar (por ejemplo, C tiene estándares ansi-c, c89, c99, etc; C++ tiene estándares C++99, c++11, c++17, etc).

Si no se puede corregir la falla, habrá que buscar en Google, indicando el nombre de la librería y pegando el mensaje de error mostrado.